OpenAPI 2.0 mukainen dokumentointi olemassa olevalle REST-rajapinnalle

Ammattikorkeakoulututkinnon opinnäytetyö

Tietotekniikan koulutusohjelma

Riihimäki, kevät 2018

Santtu Tuovinen

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Tietotekniikan koulutusohjelma
Riihimäki

Tekijä Santtu Tuovinen **Vuosi** 2018

Työn nimi OpenAPI 2.0 mukainen dokumentointi olemassa olevalle REST-rajapinnalle

Työn ohjaaja/t Toni Laitinen

TIIVISTELMÄ

Opinnäytetyön tarkoitus on toimia oppaana, kuinka jo olemassa olevalle REST-rajapinnalle saadaan luotua OpenAPI 2.0 mukainen dokumentointi. Työn alussa tutustutaan OAS 2.0 kehikkoon, sen tarjoamiin työkaluihin sekä syntaksiin.

Työn tilaaja oli Sovelluskontti Oy. Opinnäytetyön käytännön osuudessa luotiin yrityksen asiakkaan rajapintaan OpenApi 2.0 mukainen dokumentointi. Rajapinnan laajuuden vuoksi valittiin endpointeista kolme, joista yhtä käytetään esimerkkinä.

Opinnäytetyö luo pohjan, jonka perusteella voi lähteä luomaan dokumentointia olemassa olevalle REST-rajapinnalle.

Avainsanat REST-rajapinta, OAS 2.0, OpenAPI 2.0

Sivut 19 sivua, joista liitteitä 4 sivua

HÄMEEN AMMATTIKORKEAKOULU
HÄME UNIVERSITY OF APPLIED SCIENCES

Degree Program in Information Technology
Riihimäki

Author Santtu Tuovinen **Year** 2018

Subject OpenAPI 2.0 documentation for existing REST API

Supervisors Toni Laitinen

ABSTRACT

Main point of this thesis is to work as a starting guide for creating OpenAPI 2.0 documentation for existing REST API. First, we take a look to OAS 2.0 framework, syntax and tools it offers.

The Commissioner on this thesis is Sovelluskontti Oy. In practise, the created OpenAPI 2.0 documentation is for their clients REST API. Due to the size of the API three endpoints were chosen and one of them is broken down as an example.

This thesis creates a foundation which can be used to create documentation for existing REST APIs.

Keywords REST, API, OpenAPI specification, OAS 2.0

Pages 19 pages including appendices 4 pages

SISÄLLYS

1 JOHDANTO

Yksinkertaisimmillaan rajapinnat ovat kokoelma ehtoja, kuinka ohjelmat voivat keskustella keskenään. Rajapinnat voidaan suunnitella yhden ohjelman käytettäväksi tai vastaavasti ohjelma voi jakaa osia sen ominaisuuksista ilman, että valmistajan tarvitsee jakaa alkuperäistä koodia. Tiedon välitys ohjelmistojen välillä tapahtuu usein http-protokollan avulla.

REST on yksi monista vaihtoehdoista rajapintojen luomiselle ja kuluttamiselle. Tämän opinnäytetyön tarkoituksena on opastaa, kuinka ja mitä ohjelmia käyttäen voidaan jo olemassa olevalle REST-rajapinnalle luoda OpenAPI 2.0 mukainen dokumentointi ja generoida tämä toisille ohjelmointikielille. Työssä käytettävä esimerkki on luotu käyttäen PHP:n Slim-kehystä ja työkalut valittu sen perusteella.

Opinnäytetyössä pyritään vastaamaan seuraaviin kysymyksiin: Millä tavoin rajapinnan kuvaus voidaan tehdä käyttäen OpenAPI 2.0 määritelmää? Mitä työkaluja käyttäen saadaan asiakaskoodi generoitua? Tarvitseeko olemassa olevaan REST-rajapintaan tehdä muutoksia, kun se kuvataan käyttäen OpenAPI 2.0 määritelmää? Minkälaisia hyötyjä tällaisesta rajapinnan kuvauksesta on esimerkiksi liiketoiminnalle tai koodarille?

Työn tilaajana toimii Sovelluskontti Oy ja työssä käytetään esimerkkinä asiakkaan REST-rajapintaa.

2 TERMIT

OAS
OAS on lyhenne sanoista OpenAPI specification ja se tunnettiin ennen muutosta nimellä Swagger. OAS määrittää standardin ohjelmointikielestä riippumattomalle REST-rajapinnalle, jonka avulla ihmiset ja tietokoneet ymmärtävät palvelun tarkoituksen ilman lähdekoodia. (OpenApi Initiative , 2017).

OAI
OAI on lyhenne sanoista OpenAPI iniative ja se on pääorganisaatio OAS:n takana. (OpenApi Initiative 2017).

JSON
JavaScript Object Notation on ohjelmointikielestä riippumaton tiedon esitys standardi. JSON objektissa tieto esitetään kaarisulkujen sisällä avain-arvopari yhdistelmänä. Objektia voidaan kutsua avaimen nimen perusteella. (IETF 2014).

YAML
YAML Ain't Markup languagen toimintaperiaate on sama kuin JSON:ssa, mutta kaarisulkeiden sijaan käytetään sisennyksiä ja miinusmerkkejä määrittämään objektin tai listan alkamiskohta. Suurissa määrissä helpompi lukea kuin JSON. (Ben-Kiki & Evans 2009).

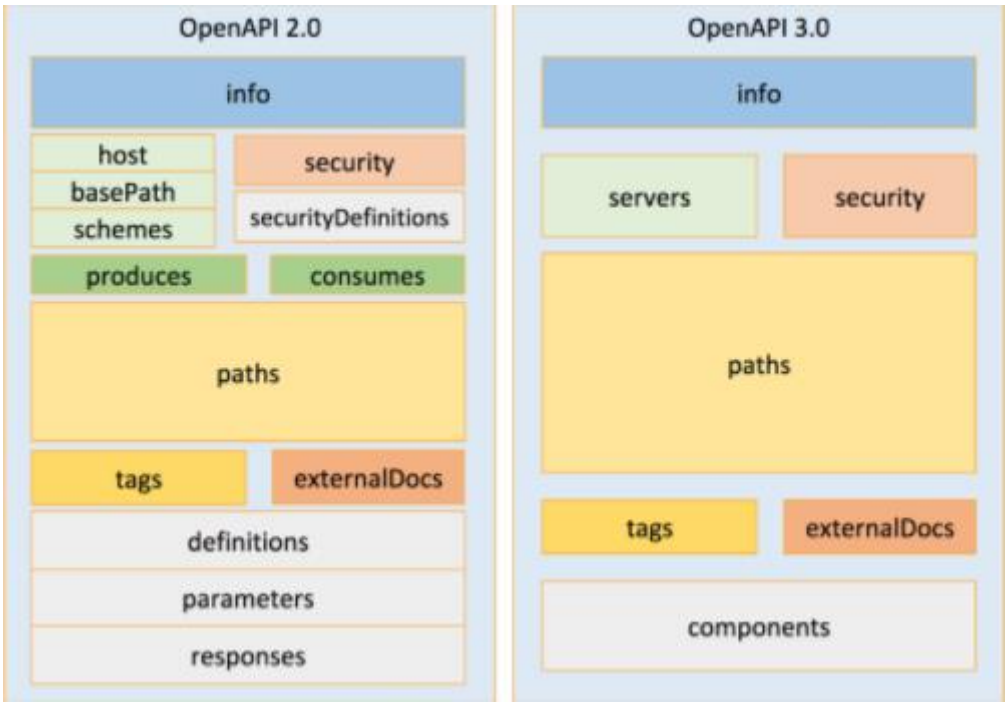
Tietotyyppi
Rajapintakuvauksen keskeinen osa on tietyyppien määrittäminen. Niiden pois jättäminen aiheuttaa viimeistään koodin generoimisessa ongelmia ja yleensä virheitä.

3 OPENAPI SPECIFICATION 2.0

Vuonna 2010 ensimmäisen Swagger-spesifikaation loi Wordnik, joka julkaisi sen vuoden kuluttua avoimen lähdekoodin alle. Maaliskuussa 2015 SmartBear osti Wordnikin osuuden Swagger-projektista emoyhtiö Raverb Technologiesilta. Vuosina 2016-2017 tapahtui nimenmuutos: Swaggerista tuli OpenAPI specification. Määritelmän takana on organisaatio, joka tunnetaan nimellä OpenAPI initiative. OAS:n perässä mainitaan yleensä myös versionumero 2.0 tai 3.0. (OpenApi Initiative 2017).

Marraskuussa 2015 SmartBear, yhdessä 3Scalen, Apigeen,Capital Onen, Googlen, IBM:n, Intuitin, Microsoftin, Paypalin ja Restletin julkisti avoimen lähdekoodin muodostamisen Linux-säätiön alaisuudessa. Osana OAI:n muodostamista SmartBear lahjoitti olemassa olevan Swagger-määrityksen Linux-säätiölle, mikä tarkoittaa, että OAS 2.0 määritelmä on sama kuin aikaisemmin tunnettussa Swaggerissa. OAS on laajasti tunnustettu olevan suosituin avoimen lähdekoodin kehys REST sovellusrajapintojen määrittelemiseksi ja luomiseksi. Nykyään kymmenet tuhannet kehittäjät rakentavat tuhansia avoimen lähdekoodin apuvälineitä, jotka hyödyntävät OpenAPI-määrityksiä. (OpenApi Initiative 2017).

Opinnäytetyön kirjoitushetkellä OAS-määrityksestä on tullut versio 3.0. Versioiden välillä suurimmat eroavaisuudet ovat kenttämäärityksissä. Versiossa 2.0 on huomattavasti enemmän objekteja verrattuna uusimpaan versioon, joka voidaan nähdä kuvasta 1.



Kuva 1. OpenAPI 2.0 vs 3.0. (OpenApi Initiative 2017).

Opinnäytetyön käytännön osuudessa oli alun perin tarkoitus käyttää OAS 3.0-määritystä, mutta työnkannalta oleellinen työkalu Swagger-php ei vielä tukenut uusinta versiota.

Olennainen osa OAS 2.0 määritelmää on editori, jonka avulla rajapintakuvaus voidaan tehdä käyttäen YAML- tai JSON-syntaksia. Swagger editorista löytyy selaimessa toimiva versio, tai ladattava UI http-sovellus. Editorin paras, tai ehkä tärkein ominaisuus on jaettu näkymä, jonka avulla työn alla oleva kuvaus nähdään reaaliajassa. Kuten suurimmassa osassa IDE:ssä, Swagger editorista löytyy telecense eli ehdotus halutusta kenttänimestä muutaman kirjaimen perusteella. Mahdollisten virheiden sattuessa, tulee niistä ilmoitus editorin ylälaitaan. Editoriin on sisäänrakennettu koodigeneraattori sekä mahdollisuus kuvauksen MOCK testaamiselle. Rajapintakuvauksen voi kirjoittaa käyttäen yksinkertaista tekstieditoria, mutta näistä ei löydy ylläkuvattuja ominaisuuksia. Kuvassa 2 on esimerkki syntaksien eroavaisuudesta YAML- ja JSON-kuvauksien välillä.

```yaml
swagger: "2.0"
info:
  description: "API esimerkki"
  version: "1.0.0"
  title: "Esimerkki API"
schemes:
  -"http"
host: "esimerkki.fi"
basePath: "/api"
```

```json
{
    "swagger": "2.0",
    "info": {
        "description": "API esimerkki",
        "vesion": "1.0.0",
        "title": "Esimerkki API",
    },
    "schemes": ["http"],
    "host": "esimerkki.fi",
    "basePath": "/api"
}
```

Kuva 2. Syntaksi YAML vs JSON. (Kuvakaappaus)

3.1 **Tietotyypit**

Taulukko 1. OAS 2.0-tietotyypit (OpenAPI Specification 2.0 2018).

Yleisnimi	**Kuvaustyyppi**	**Formaatti**
integer	integer	int32
long	integer	int64
float	number	float
double	number	double
string	string	
byte	string	byte
binary	string	binary
boolean	boolean	
date	string	date
dateTime	string	date-time
password	string	password

Taulukossa 1 on listattuna OAS 2.0 määrityksen tukemat tietotyypit. Yleisnimi sarakkeesta löytyvät tietotyyppitermit ovat käytössä myös muissa ohjelmointikielissä. Kuvaustyyppi sarakkeessa on kuvauskielessä käytetty vastine. Viimeisestä sarakkeesta löytyvät formaatit, joiden avulla voidaan tarkentaa tietotyyppejä. Jos rajapintakuvauksen pohjalta generoidaan lähdekoodia, on tietotyyppien määrittäminen pakollista. Muussa tapauksessa tietotyypit voidaan jättää merkitsemättä, mutta nämä kuitenkin lisäävät luettavuutta kuvauksen loppukäyttäjien kohdalla. (OpenAPI Specification 2.0 2018).

Taulukosta löytyvien tietotyyppien lisäksi OAS 2.0 antaa mahdollisuuden käyttää omia tietotyyppejä. Tätä ominaisuutta käytettäessä tietotyypille annetaan arvoksi ”string”, jonka jälkeen kuvaustyyppi sarakkeesta voidaan valita sopiva vaihtoehto ja formaatti. (OpenAPI Specification 2.0 2018).

3.2 **Toimintaperiaate**

Taulukko 2. OAS 2.0-kenttämääritykset. (OpenAPI Specification 2.0 2018)

Kenttänimi	**Kuvaustyyppi**	**Kuvaus**
swagger	string	Määrittää OAS-versionumeron, vaihtoehtoina 2.0 tai 3.0.
info	tieto objekti	Välittää rajapintakuvauksen metadatan.
version	string	Polku objektin sisällä. Ilmoittaa rajapintakuvauksen version.

title	string	Polku objektin sisällä. Määrittää rajapintakuvauksen nimen
paths	Polku objekti	Rajapintakuvauksen polut ja operaatiot.
/polunnimi	string	Suhteellinen polku yksilölliseen endpointtiin.
get, put, post, delete, options, head, patch	Toiminto objekti	Haluttu toiminto kuvauksessa.
parameters	Parametri objekti	Lista parametreistä jotka soveltuvat toimintoon
- name	string	Muuttujan nimi, parametri objektin sisällä
in	string	Parametrin sijainti. Mahdolliset arvot ovat: query, header, path, formData ja body
type	string	Muuttujan tietotyyppi, lueteltu Taulukossa 1
responses	Vastaus objekti	Lista mahdollisista vastauksista operaation suorittamisesta
http-statuskoodi numero		Esimerkiksi 200, 400 ja 404
description	string	Vastaus objektin sisällä. Lyhyt kuvaus vastauksesta

OAS 2.0 pitää sisällään valtavan määrän kenttämäärityksiä, objekteja ja muita arvoja. Tämän johdosta taulukkoon 2 on merkitty kaikki kuvauskielen pakolliset kentät. Muita OAS 2.0 tarjoamia kenttiä voi tarkastella osoitteesta https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md. Ensimmäisten neljän rivin avulla saadaan aikaan toimiva metadata rajapintakuvaukselle. Polku objektin alle kuvataan kaikki tarvittavat toiminnot ja sen sisältämät parametrit. Vastaus objektin sisälle tulee kaikki toiminnon mahdollisesti palauttavat statuskoodit. Toimiva rajapintakuvaus voi lyhimmillään olla noin 15 riviä pitkä.

3.3 **Viittausmääritykset**

Rajapintakuvauksien lyhentämiseksi ja luettavuuden lisäämiseksi OAS 2.0-kuvauskielessä on mahdollisuus globaalien tietotyyppimäärityksien tekemiselle. Ideaalitilanteessa samaa kuvausta voidaan käyttää useammassa kohtaa. Globaalit kuvaukset voidaan lisätä dokumentin loppuun definitions kenttänimen alle. Esimerkki tällaisesta löytyy kuvasta 3.

```
43   definitions:
44     Example:
45       type: "object"
46       properties:
47         example_id:
48           type: "integer"
49           format: "int64"
50         quantity:
51           type: "integer"
52           format: "int32"
53         date:
54           type: "string"
55           format: "date-time"
56         status:
57           type: "string"
58           description: "Esimerkki"
```

Kuva 3. Esimerkki viittausmäärityksestä. (Kuvakaappaus)

Kun kuvaus on valmis, voidaan siihen viitata halutussa kohdassa. Viittaus kuvasta löytyvään esimerkkiin olisi seuraavanlainen: $ref:”#/definitions/Example.

```
paths:
  /esim:
    get:
      summary: "Get all data"
      responses:
        200:
          description: "OK"
          schema:
            $ref: "#/definitions/Esim"
        401:
          $ref: "#/responses/401Err"


definitions:
  Esim:
    type: "object"
    properties:
      esim1:
        type: "string"
      esim2:
        type: "integer"

  401:
    type: "object"
    properties:
      esim_id:
        type: "integer"
      esim_firstname:
        type: "string"

responses:
  401Err:
    description: "Not Authorized"
    schema:
      $ref: "#/definitions/401"
```

Kuva 4. Monimutkaisempi viittausmääritys. (Kuvakaappaus)

Esimerkki hieman monimutkaisemmasta rakenteesta löytyy kuvasta 4. Kun palvelin palauttaa statuskoodin 200, tulee tietosisältö globaalin tietotyyppikuvauksen ”Esim” mukaisesti. Palvelimen palauttaessa 401, on ongelma tunnistuksen kanssa. Kuten esimerkistä huomataan, viittauksia voi käyttää hyvin monipuolisesti.

4 TYÖKALUT

4.1 Swagger-php

Swagger-php on avoimen lähdekoodin työkalu, jolla saadaan luotua dokumentointi jo valmiina olevaan REST-rajapintaan. Tämä tapahtuu lisäämällä kommentteja tiedostoon Doctrinen määrittämällä tyylillä. Valmiina olevat endpointit voidaan kuvata uudestaan samaan tiedostoon rikkomatta alkuperäistä ja toimivaa tiedostoa. Kun kaikki kuvaukset ovat valmiina, voidaan komentoriviltä ajaa komento, joka luo kommenttien pohjalta JSON tiedoston. Työkalun valinnan pohjalla oli sen OAS 2.0 tuki, sekä samanlaisuus kyseisen version syntaksin kanssa.

4.2 Swagger Code Generator

Koodigeneraattorin avulla voidaan edellisellä työkalulla saatu JSON tiedosto kääntää halutulle kielelle. Koodigeneraattoria voidaan käyttää paikallisesti tai editorin avulla. Paikallisesti toimivan voi ladata https://github.com/swagger-api/swagger-codegen osoitteesta. Paikallisesti toimiva koodigeneraattori tarvitsee lisäksi Java-SDK 7 tai uudemman ja Apachen-mavenin toimiakseen. Koska generaattori on asennettu, voidaan sille antaa generoinnin yhteydessä parametrejä paremman ja tarkemman lopputuloksen aikaansaamiseksi.

Swagger editorissa on myös mahdollisuus koodin generoimiseen, mutta toisin kuin paikallisessa versiossa, ainoa määritettävä asia on ohjelmointikieli. Tuetut ohjelmointikielet palvelimille (Generate server):
Ada, C# (ASP.NET Core, NancyFx), C++ (Pistache, Restbed), Erlang, Go, Haskell (Servant), Java (MSF4J, Spring, Undertow, JAX-RS: CDI, CFX, Inflector, RestEasy, Play Framework, PKMST), Kotlin, PHP (Lumen, Slim, Silex, Symfony, Zend Expressive), Python (Flask), NodeJS, Ruby (Sinatra, Rails5), Rust (rust-server), Scala (Finch, Lagom, Scalatra). (Community 2018).

Tuetut asiakas ohjelmointikielet (Generate Client) ovat:
ActionScript, Ada, Apex, Bash, C# (.net 2.0, 3.5 tai uudempi), C++ (cpprest, Qt5, Tizen), Clojure, Dart, Elixir, Elm, Eiffel, Erlang, Go, Groovy, Haskell(http-client, Servant), Java (Jersey1.x, Jersey2.x, OkHttp, Retrofit1.x, Retrofit2.x, Feign, RestTemplate, RESTEasy, Vertx, Google API Client Libary for Java, Rest-assured), Kotlin, Lua, Node.js (ES5, ES6, AngularJS with google Closure Compiler annotations) Objective-C, Perl, PHP, PowerShell, Python, R, Ruby, Rust, (rust, rust-server), Scala (akka, http4s, swagger-async-httpclient), Swift (2.x, 3.x, 4.x), TypeScript (Angular1.x, Angular2.x, Fetch, jQuery, Node). (Community 2018).

Koodigeneraattori muodostaa lähteestä paketin, joka sisältää rajapintakuvauksen polut, tietotyyppikuvaukset, kontrollerit jne. Kansiosta löytyvästä

readme-tiedostosta löytyvät ohjeet ja tarvittavat kirjastot paketin asentamista varten. Ohjelmointikielien määrästä riippuen paketin käytössä ja asennuksessa saattaa esiintyä eroavaisuuksia. (Community 2018).

4.3 **Postman**

Postman on monitoimityökalu, jolla voi suunnitella, testata, dokumentoida ja julkaista rajapintakuvauksia. Ohjelmasta löytyvä kokoelmaominaisuus on erittäin hyödyllinen. Sen avulla ohjelmaan voidaan ladata JSON päätteinen tiedosto, joka luo automaattisesti kokoelman rajapinnan endpointeista. Tämän jälkeen listasta voidaan valita haluttu polku testausta varten ja määrittää halutut parametrit (header, body tai erilaiset testit). (Postman 2018).

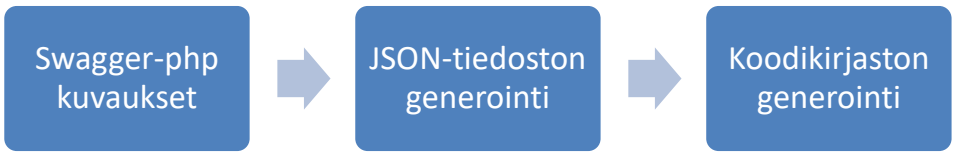
5 TOTEUTUS

Opinnäytetyön käytännön osuudessa luotiin dokumentointi jo olemassa olevalle REST-rajapinnalle. Koska OpenAPIn työkaluihin kuuluu koodigeneraattori, dokumentoinnin tarkoituksena on jatkossa lyhentää ohjelmoijien työmäärää. Asiakkaalla on käytössä websovelluksen lisäksi natiivit Android- ja iOS-sovellukset, joten muutokset voisi tehdä vain yhteen dokumenttiin. Sovellus toimii lähes kokonaan POST-metodeilla, joista valitsin yhden esimerkiksi.

Työn alussa asensin yritykseltä saatujen ohjeiden mukaan testiympäristön. Tämä oli toteutettu Dockerin avulla, johon loin ohjeiden mukaisesti kaksi konttia. Ensimmäisenä oli mySQL, johon ladattiin myös tyhjä kopio tietokannasta. Toisena konttina toimi Ubuntu, johon asennettu Apache mahdollisti paikallisen kehityksen. Lopuksi kontit linkitettiin yhteen tiedonkulun mahdollistamiseksi. Dockerfile, sekä asennukseen käytetyt komennot löytyvät liitteistä 1 ja 2.

Työn prosessi on kuvattu kuviossa 1.

Kuvio 1. Dokumentointi prosessi.



5.1 Kuvauksien lisääminen

Prosessi alkaa käyttäen swagger-php työkalua. Työkalun asentaminen edellyttää, että tietokoneelta löytyy Composer. Asentaminen tapahtuu komennolla: composer global require zircote/swagger-php. Swagger-php on tämän jälkeen asennettu globaalisti ja sitä voidaan käyttää komentoriviltä liikkumatta ensin asennuskansioon. Swagger-php:n avulla haluttuun dokumenttiin voidaan lisätä kuvaukset ilman, että se rikkoo alkuperäistä koodia. Swagger-php toimii lisäämällä kommentteja Doctrinen määrittämällä tavalla. Tämä toimii seuraavasti: Kaikki kuvaukset kirjoitetaan

kommenttikentän sisälle. Parametrit tai metodit aloitetaan @SWG\ ja lisäämällä haluttu toiminto.

Kuten kuvasta 5 huomataan, tehdyt kommentit noudattavat OpenAPI 2 määritelmää. Muutamia eroavaisuuksia kuitenkin on. Swagger-php tukee listoja, mutta hakasulkeiden sijasta se käyttää kaarisulkeita, lisäksi objektit merkataan käyttäen normaaleita sulkeita. Myös metodien nimet eroavat toisistaan tietyissä tilanteissa, esimerkiksi parameters on muutettu yksikkömuotoon parameter.

Kuvassa 5 on esimerkki tiedoista, jotka OpenAPI 2 määritys tarvitsee toimiakseen. Kuvassa näkyvien tietojen lisäksi infokenttään voidaan halutessa lisätä webosoitteita, lisenssi- sekä yhteystietoja. Tämä osuus kannattaa pitää dokumentin alussa selkeyden vuoksi. Asiakkaan nimen suojaamiseksi tiedot on vaihdettu keksittyyn esimerkkiin.

```
/**
 * @SWG\Swagger(
 *  @SWG\Info(
 *    description="API esimerkki",
 *    vesion="1.0.0",
 *    title="Esimerkki API"
 *  ),
 *  schemes={"http"},
 *  host="esimerkki.fi",
 *  basePath="/api"
 *)
 */
```

```
{
    "swagger": "2.0",
    "info": {
        "description": "API esimerkki",
        "vesion": "1.0.0",
        "title": "Esimerkki API",
    },
    "schemes": ["http"],
    "host": "esimerkki.fi",
    "basePath": "/api"
}
```

Kuva 5. Swagger-php kuvaus vs JSON. (Kuvakaappaus)

Kun tarvittavat tiedot on lisätty tiedoston alkuun, voidaan aloittaa metodien kuvaaminen. Esimerkkinä toimii POST-metodi, jonka tarkoituksena on lähettää tieto työvuoron aloittamisesta. Alkuperäinen koodi löytyy liitteestä 3. Polku määrittää halutun endpointin polun, ”summary” on tieto, jolla kuvaus näkyy esimerkiksi Postmanissa. Tagien avulla kuvaukset voidaan halutessa jakaa endpointit otsikoiden alle. Ennen parametrejä määritetään vielä, missä muodossa palautetut tiedot halutaan.

Kuvassa 6 on POST-metodin ensimmäinen parametri. Nimi määrittää yksiselitteisesti parametrin nimen ja in missä tiedon halutaan kulkevan. Näiden lisäksi on pakollista määrittää myös tietotyyppi. Ilman tietotyypin määritystä viimeistään koodin generointi vaiheessa törmätään virheisiin. Parametrin tieto on lopuksi määritetty pakolliseksi.

```
/**
 * @SWG\Post(
 *   path="/api/start/shift",
 *   summary="/api/start/shift",
 *   operationId="startShift",
 *   tags={"duunissa"},
 *   produces={"application/json"},
 *   @SWG\Parameter(
 *     name="Authorization",
 *     in="header",
 *     type="string",
 *     description="AuthToken",
 *     required=true
 *   ),
```

Kuva 6. Post-metodin ylätunnus (Kuvakaappaus)

Seuraavaksi vuorossa on Post-metodin bodyn kuvaaminen. Kentän pakollisuus on asetettu epätodeksi, koska mukana menevät parametrit eivät ole pakollisia. Scheman sisällä oleva otsikkokenttä määrittää millä nimellä luokka esiintyy, kun se generoidaan. Jokainen arvo pitää merkitä erikseen ja niiden tulee sisältää kaksi pakollista kenttää, ominaisuus ja tietotyyppi.

```
 *   ),
 *   @SWG\Parameter(
 *     name="body",
 *     in="body",
 *     description="Posts timestamp that is used to start shift",
 *     required=false,
 *     @SWG\Schema(
 *       title="startShiftBody",
 *       @SWG\Property(
 *         property="ho_lat",
 *         type="string"
 *       ),
 *       @SWG\Property(
 *         property="ho_lon",
 *         type="string"
 *       ),
 *       @SWG\Property(
 *         property="ho_address",
 *         type="string"
 *       )
 *     )
 *   ),
```

Kuva 7. Post-metodin body-osuus. (Kuvakaappaus)

Viimeisenä määritetään mahdolliset vastaukset. Kyseisen metodin kohdalla näitä on kaksi. Ensimmäisenä 200, joka tarkoittaa kaiken onnistuneen, toisena 401 joka viittaa tunnistuksen epäonnistuneen.

OpenAPI 2 tavoin swagger-php tukee viittauksia, jota käytin vastauksen 200 kohdalla. Tässä tärkeimpänä on huomata ensimmäisenä dollarin merkki, jonka avulla saadaan vastauksesta luotua oma luokka generoinnin aikana.

```
135     *  @SWG\Response(
136     *    response=200,
137     *    description="Succesfully stared new shift",
138     *    ref="$/responses/Json_startShift"
139     *  ),
140     *  @SWG\Response(
141     *    response=401,
142     *    description="Not authorized"
143     *  )
144     *
145     * )
146     */
```

Kuva 8. Post-metodin palauttamat http-statuskoodit. (Kuvakaappaus)

Kuvassa 9 on kuvattu referoitu 200 vastaus. Vastaus-kenttä määrittää kuvaukselle nimen, jonka avulla siihen voidaan viitata. Otsikko-kenttä on generoinnissa syntyvän luokan nimi. Tämän jälkeen palautuvat kentät voidaan määrittää tietotyyppeineen.

```
149     /**
150     *     @SWG\Response(
151     *       response="Json_startShift",
152     *       description="Json response",
153     *       @SWG\Schema(
154     *         title="startShiftResponse",
155     *         @SWG\Property(
156     *           property="message",
157     *           type="string"
158     *           ),
159     *         @SWG\Property(
160     *           property="action",
161     *           type="integer"
162     *           ),
163     *         @SWG\Property(
164     *           property="success",
165     *           type="boolean"
166     *           ),
167     *         @SWG\Property(
168     *           property="error_message",
169     *           type="string"
170     *           ),
171     *       )
172     *     )
173      */
```

Kuva 9. Viittausmääritys 200-statuskoodille.

Kun kaikki halutut metodit on kuvattu, voidaan swagger-php:lle antaa komentoriviltä seuraava käsky: swagger D:\Hamk\opari\opari\opari\api\src --output D:\Hamk\opari\opari\json. Komennon parametreinä on projektikansion sijainti sekä polku generoidun tiedoston sijoittamiselle. Tämän jälkeen ohjelma käy läpi kaikki tiedostot kansiossa. Mahdollisen virheen sattuessa käyttäjä saa tietoonsa ainoastaan metodin alkamiskohdan.

5.2 **Testaus**

Koska generoitu tiedosto on JSON-päätteinen, voidaan se ladata suoraan Postmanin kokoelmaksi. Postman on loistava työkalu, jonka avulla jokainen kuvaus voidaan testata erikseen. Käyttäjä voi halutessaan lisätä / muuttaa tietoja, joita metodi kuljettaa mukanaan.

Kun kuvaus on todettu toimivaksi, voidaan sitä lähteä kääntämään toiselle ohjelmointikielelle. Tämä prosessi tapahtuu Swagger Code Generatorin

avulla. Koska kaikki työ on tehty tähän mennessä paikallisesti, ladattiin kyseisestä ohjelmasta paikallinen versio.
Ensin komentorivillä pitää navigoida ladattuun swagger-codegen kansioon. Tämän jälkeen voidaan suorittaa seuraava komento: java -jar modules\swagger-codegen-cli\target\swagger-codegen-cli.jar generate -i D:\Hamk\opari\opari\json\swagger.json -l php -o c:\temp\php_api_client. Ajettavan komennon parametreinä on generoitava tiedosto, haluttu asiakas- tai palvelinkieli sekä kohdekansio. Luodusta kansiosta löytyvästä README.md tiedostosta löytyy tämän jälkeen ohjeet kyseisen paketin asentamiselle ja testaamiselle.

Aiemmin tehdyn kuvauksen ja luodun paikallisen testiympäristön kanssa oli yhteensopivuusongelmia, joten käytin viimeiseen esimerkkiin valmista kuvausta, joka löytyi swagger editorista. Generoin tämän pohjalta itselleni JavaScript kirjaston. Noudattamalla asennusohjeita, suoritin kuvasta 10 löytyvän koodin komentoriviltä node-moduulin avulla.

```
1
2   var UberApi = require('uber_api');
3   var api = new UberApi.EstimatesApi()
4
5   var startLatitude = 1.2; // {Number} Latitude component of start location.
6
7   var startLongitude = 1.2; // {Number} Longitude component of start location.
8
9   var endLatitude = 1.2; // {Number} Latitude component of end location.
10
11  var endLongitude = 1.2; // {Number} Longitude component of end location.
12
13
14  var callback = function(error, data, response) {
15    if (error) {
16      console.error(error);
17    } else {
18      console.log('API called successfully. Returned data: ' + data);
19    }
20  };
21  api.estimatesPriceGet(startLatitude, startLongitude, endLatitude, endLongitude, callback);
```

Kuva 10. Node-moduulin suorittama koodi. (Kuvakaappaus)

Kuvasta 11 löytyy pieni osa palautetusta datasta.

```
reg:
  ClientRequest {
    domain: null,
    _events: [Object],
    _eventsCount: 3,
    _maxListeners: undefined,
    output: [],
    outputEncodings: [],
    outputCallbacks: [],
    outputsize: 0,
    writable: true,
    _last: true,
    upgrading: false,
    chunckEncoding: false,
    shouldKeepAlive: false,
    useChunkedEncodingByDefault: false,
    sendDate: false
  }
```

Kuva 11. Palautettu data. (Kuvakaappaus)

Työn aikana kuvattiin kolme endpointtia, jotka olivat tunnistautuminen, työvuoron lopettaminen ja työvuoron aloittaminen, joka löytyy liitteestä 3. Metodeihin tehdyt kuvaukset olivat rakenteeltaan lähes identtiset ja ainoat erot löytyivät palautetusta datasta. Työn alussa oli esitetty kysymys: tarvitseeko olemassa olevaan REST-rajapintaan tehdä muutoksia, kun se kuvataan käyttäen OpenAPI 2.0 määritelmää? Olemassa oleva rajapinta on kuitenkin hyvin laaja ja testissä oli kolme metodia. Näiden kohdalla muutoksia alkuperäiseen koodiin ei tarvinnut tehdä.

6 YHTEENVETO

Opinnäytetyön tarkoituksena oli luoda opas, kuinka valmiille REST-rajapinnalle voidaan luoda OAS 2.0 mukainen dokumentointi. Dokumentin alussa tutustuttiin ensin OAS 2.0 kuvauskieleen, sen syntaksiin ja käytiin läpi tarjottuja työkaluja. Vaikka työn aikana ei kyseisiä työkaluja käytetty, on valmiin rajapintakuvauksen päälle tehtävän dokumentoinnin kannalta tärkeää ymmärtää OAS 2.0 rakenne. Opinnäytetyön käytännön vaiheessa luotiin dokumentointi Sovelluskontti Oy:n asiakkaan rajapinnalle ja käytiin läpi työkaluja sekä prosessia.

Alussa esitettyyn kysymykseen, mitä hyötyjä edellä tehdyllä kuvauksella on, ei ole yksiselitteistä vastausta. Alkuperäisen ja jatkuvan kehityksen johdosta rajapintakuvaus on varsin pitkä ja jatkaa kasvamistaan. Tämän johdosta dokumenttiin tehtävä uusi kuvaus veisi varmasti paljon aikaa ja olisi työläs toteuttaa. Kyseisen sovelluksen kohdalla on websovelluksen lisäksi olemassa natiivit Android- ja iOS-sovellukset. Tämä tarkoittaa siis sitä, että rajapintakuvausta laajennettaessa, muutokset joudutaan tekemään manuaalisesti kolmella eri ohjelmointikielellä. Pitkällä aikavälillä dokumentoinnin kuvaaminen uudestaan voisi vähentää tarvittavaa työtä jatkokehityksen kannalta.

LÄHTEET

Ben-Kiki, O.;& Evans, C. (2009). YAML. Haettu 27.2.2018 osoitteesta http://www.yaml.org/about.html

Community (2018). gitHub. Haettu 22.3.2018 osoitteesta https://github.com/swagger-api/swagger-codegen

IETF (2014). IETF. Haettu 27.2.2018 osoitteesta https://tools.ietf.org/html/rfc7159

OpenApi Initiative (2017). OpenApi Initiative. Haettu 27.2.2018 osoitteesta https://www.openapis.org/faq

OpenAPI Specification 2.0 (2018). Haettu 4.4.2018 osoitteesta https://github.com/OAI/OpenAPI-Specification/blob/master/versions/2.0.md

Postman (2018). Haettu 22. 3. 2018 osoitteesta https://www.getpostman.com/

Liite 1

Dockerfile

```
######################
# Updated: 9.1.2017 #
######################


# CHANGELOG:
#
# 9.1.2017:
# * Upgraded php 5.5.x repo + packages to version 5.6
# * Added more awesome
# * New table added to DB -- .sql file in docker folder!

# Pull base image
FROM ubuntu:14.04

ENV TERM xterm

# Add runtime ARG for build (ENV persistaa jos tästä periyttää jtn)
ARG DEBIAN_FRONTEND=noninteractive

# Set locales
# add-apt-repositorya varten kun ppa:ondrej/php kanssa tulee jotain on-
gelmia encoodauksien kanssa.
RUN locale-gen en_US.UTF-8
RUN export LANG=en_US.UTF-8
RUN export LC_ALL=en_US.UTF-8

# install to enable add-apt-repository as per 14.04 guidelines
# Listään repo ja sen jälkeen päivitetään listaus niin apt-get install komento
toimii

RUN apt-get update && apt-get install -y  \
  software-properties-common \
  python-software-properties \
 && add-apt-repository ppa:ondrej/php

# Install software
# Something wants --force-yes so tech perhaps?

RUN apt-get update && apt-get install -y --force-yes \

  apache2 \
  vim \
  nano \
```

```
  php5.6 \
  php5.6-mcrypt \
  php5.6-mbstring \
  php5.6-curl \
  php5.6-cli \
  php5.6-mysql \
  php5.6-gd \
  php5.6-intl \
  php5.6-xsl \
  php5.6-zip \
# Poistetaan cachet
&& apt-get clean \
&& rm -rf /var/lib/apt/lists/*

# apache2 vhosts before ensites.
# enable rewrite module
RUN a2enmod rewrite
# enable ssl module
RUN a2enmod ssl
# create directory for ssl certificates
RUN mkdir /etc/apache2/ssl
# create ssl certificates
RUN openssl req -x509 -nodes -days 365 -newkey rsa:2048 -keyout
/etc/apache2/ssl/apache.key -out /etc/apache2/ssl/apache.crt -subj
"/C=FI/ST=Hameenlinna/L=Hameenlinna/O=Sovelluskontti/OU=Develop
ment/CN=opari.local"
# copy virtualhost configurations
COPY ["virtualhosts/*.conf", "/etc/apache2/sites-available/"]
# enable virtualhosts
RUN a2ensite local-opari.conf
RUN a2ensite local-opari-rakennus.conf
RUN a2ensite default-ssl.conf

# Määritetään apachen käyttävän php5.6 versiota.
# Ensin disabloidaan php5 ja sitten otetaan uusi käyttöön.
# Restartin aikana dokcer yhteys voi katketa, joten docker apache pitää
käynnistää uudelleen.
# Kommentoi eka rivi pois, jos php5 modulia ei löydy

# try to disable php5 if the module exists
# This will always return a 0 (success) exit code.

 RUN a2dismod php5; exit 0
 RUN a2enmod php5.6


 RUN echo Europe/Helsinki | sudo tee /etc/timezone && sudo dpkg-recon-
figure --frontend noninteractive tzdata
```

```
# php komennon kuuluisi tulostaa, että 5.6 käytössä tai jokin versionu-
mero.
# Voit tarkistaa, että apache käyttää oikeaa php -versiota kun menet osoit-
teeseen opari.local:8080/phpinfo.php

 RUN php -v

RUN service apache2 restart

EXPOSE  80

CMD ["/usr/sbin/apache2ctl", "-D", "FOREGROUND"]
```

Liite 2

Docker komennot

```
# Pull mysql image from dockerhub

docker pull mysql:5.6

# check that image is downloaded

docker images

# create mysql container

docker run -p 3900:3306 --name mysql-opari -e MYSQL_ROOT_PASSWORD=root -d mysql:5.6

# verify that container is running

docker ps

# connect to mysql container

docker exec -it mysql-opari bash

# inside container, run next command

echo Europe/Helsinki | tee /etc/timezone && dpkg-reconfigure --frontend noninteractive tzdata

# then exit from container

exit

docker build -t opari/local:apache-opari .\docker\.

# when build is finished, verify that you see image (apache-opari)

docker images


docker run -d -p 8080:80 -p 443:443 --name apache-opari --link mysql-opari:mysql -v //c/dev/opari:/var/www/html -v //c/dev/opari/apache2:/var/run/apache2 opari/local:apache-opari
```

Liite 3

Koodiesimerkki

```
$app->post('/start/shift[/{:offline}]', function($request, $response, $args) use ($app) {

		global $Projects;

		$errors = array();

		$userdata = $response->getHeader('userdata');

		if (isset($args['offline']) && $args['offline'] == 'offline') {
			$offline = 1;
		} else {
			$offline = 0;
		}

// #238:  ios / android
 $device = $request->getParam('device') ? $request->getParam('device') : NULL;
 $version = $request->getParam('version') ? $request->getParam('version') : NULL;

 $deviceObject = new AuditDevice($device, $version);

 $ho_team = explode(",", $request->getParam('ho_team'));
		$ho_lat = $request->getParam('ho_lat');
		$ho_lon = $request->getParam('ho_lon');
		$ho_address = $request->getParam('ho_address');
		$ho_project_new_name = $request->getParam('ho_project_new_name');
		$ho_project_id = $request->getParam('ho_project_id');
		$ho_offline_timestamp = $request->getParam('ho_offline_timestamp');


// Start a shift
		$output = $app->dsApi->startShift(
			 $userdata['user_id']
			,$ho_project_id
			,''
			,$ho_team
			,$ho_lat
			,$ho_lon
			,$ho_address
			,$offline
			,$ho_offline_timestamp
```

```
        ,$deviceObject
                    );

                    return $response->withStatus(200)->withJson([
                                'message' => 'authorized'
                                ,'action' => $output['action']
                                ,'success' => $output['success']
                                ,'error_message' => $output['error_message']
                    ]);

})->add($authenticate);
```